Towards a secure Kerberos key exchange with smart cards*

Nikos Mavrogiannopoulos, Andreas Pashalidis, and Bart Preneel

Dept. of Electrical Engineering/COSIC
Katholieke Universiteit Leuven – iMinds
Belgium

July 2013

Abstract

Public key Kerberos (PKINIT) is a standard authentication and key establishment protocol. Unfortunately, it suffers from a security flaw when combined with smart cards. In particular, temporary access to a user's card enables an adversary to impersonate that user for an indefinite period of time, even after the adversary's access to the card is revoked. In this paper, we extend Shoup's key exchange security model to the smart card setting, and examine PKINIT in this model. Using this formalization, we show that PKINIT is indeed flawed, propose a fix, and provide a proof that this fix leads to a secure protocol.

1 Introduction

It is well known that human users can be authenticated based on something they know (e.g. a password), something they have (e.g. a smart card), or some part of themselves (e.g. a fingerprint). Unfortunately, possession-based authentication systems are more complex and costly to implement, deploy, and maintain than knowledge-based authentication systems. This is because, while the user's mind operates as the secure storage place for his password in a knowledge-based system, in a possession-based system hardware tokens and special software must be manufactured and managed. Analyzing the security of systems that are based on hardware tokens is also more challenging because security-critical functions are performed both by the user's terminal and the token itself (see [24] for a detailed discussion of the implications of this situation). Moreover, hardware tokens are often protected by a Personal Identification Number (PIN); while this

*This is the author's version of the paper in International Journal of Information Security with DOI 10.1007/s10207-013-0213-x.

potentially leads to stronger security, often called 'two-factor authentication', it complicates the security analysis further.

Another complicating factor, which is often overlooked, is that users may share their tokens just as they share their passwords, even if explicitly told to not do so [26]. We assume the following mental model with respect to sharing. While a user will change his password in order to revoke the rights from those with whom the password was shared, a user that wishes to revoke the rights from those with whom his *token* was shared, simply retrieves his token and keeps it at a safe place. Possession-based authentication system propagate exactly this mental model, and, therefore, must guarantee that non-possession of the token leads to the inability to authenticate.

A possession-based authentication system should also take into account the threat of an adversary that compromises the user's terminal. Assuming that the token communicates through the terminal, such a compromise affects all users, irrespective of whether or not they share their tokens with others. A compromised terminal, in addition from capturing the user's PIN, may engage in a large number of illegitimate communications with the token. Unless the token has some mechanism to alarm the user, such an attack cannot be detected.

In addition to authenticating users, token-based authentication systems are also used to establish session keys between the user's terminal and a remote server. These keys, which are used to protect the communication between the user and the server, must be secret, even to someone who had previously access to the token. The system must therefore protect these keys from threats they are exposed to whenever the token is within an adversarial environment (e.g. an adversary temporarily 'borrowed' the token, or compromised the user's terminal).

In order to address the above threats, we argued in [19] about the need of certain protocol properties for token-based authentication and key establishment systems. The following two properties ensure that this temporary compromise, does not lead to the ability for an adversary to impersonate users or servers outside the time window of the compromise.

- *SC*[1] *key-compromise impersonation:* An adversary that accessed a user's token, should not be able to impersonate other entities to that user, as long as the user uses a non-corrupted terminal.
- *Possession-based authentication:* An adversary with access to a user's token, should be able to impersonate that user only for as long as it has access to the token.

The following two properties ensure the secrecy of past and future sessions.

- *SC perfect forward secrecy:* Session keys that were established with a user's token over a non-corrupted terminal remain secret, even if an adversary later obtains access to that user's token.

[1]The SC acronym can be interpreted as smart card.

- *SC backward secrecy:* Session keys that are established with a user's token over a non-corrupted terminal remain secret, even if an adversary had previously accessed the user's token.

These properties complement the list of the desirable properties for secure communications protocol put forward in [7].

The focus of this paper is the possession-based authentication system that arises when using the Diffie-Hellman (DH) variant of public key Kerberos [30] with smart cards. While there are various types of smart cards, we consider unclonable smart cards without a keypad that provide operations on their stored keys without exposing them. That is, in our model the adversary is unable to extract long-term secrets from the card.

The informal examination in [19] shows that, unfortunately, this variant of Kerberos does not provide the property of possession-based authentication; an adversary that has temporary access to a user's smart card at some point in time, can impersonate that user for an indefinite period of time. In this paper, we formalize this work. In particular, we extend Shoup's key exchange security model [27] to the smart card setting, and examine PKINIT in this model. Based on this, we show that PKINIT is indeed flawed and provide a proof that a similar to the proposed fix in [19] leads to a secure protocol.

The rest of this paper is organized as follows. The next section overviews related work and Section 3 revisits the PKINIT protocol. The attack and a fix is described in Section 4. Section 5 revisits Shoup's static corruptions model [27] and presents our extended version of the model. This extended version accounts for the presence of smart cards in the context of key establishment protocols. Section 6 applies the model to the original and the fixed PKINIT protocols. Finally, Section 7 concludes.

2 Related work

Several models, such as the BAN logic [9, 17], process calculus-based logics [13], and complexity theory analysis techniques (e.g. [4]) capture protocol security guarantees on authentication and key exchange. The adversary they consider is typically a Dolev-Yao-style adversary [16], i.e., is assumed to control the entire network. The PKINIT protocol, including its Diffie-Hellman variant, has already been proven secure under that adversary model [2, 8, 12, 23].

However, the Dolev-Yao model does not consider adversaries with attack abilities such as temporary access to a user's card. That is, the models used for the PKINIT protocol verification do not capture attacks that involve temporary smart card access. More generally, their positive results of protocol verification do not carry over to the smart card setting.

Despite the fact that smart cards have been introduced to a multitude of protocols, to the best of our knowledge, only few works provide formal treatments of smart card-based protocols against an adversary that is able to attack a user's terminal independently from his smart card (see, for example, [3, 10, 11, 28]). In the following paragraphs we provide an overview of the existing methods and their limitations.

2.1 BAN logic

Abadi et al. enhanced BAN logic [9] to account for smart card related threats in [10]. The BAN logic is a logic of beliefs that is used to prove certain properties in authentication protocols. The smart card extension is the first known formal treatment of smart card threats written in time where the notion of a smart card was not clearly defined. As such the protocols that are studied vary from smart cards that resemble the features of a modern smart card, to smart cards that include batteries (to maintain a clock), screen and keypad. The adversaries they consider are the typical Dolev-Yao network adversary, but include the abilities for smart card theft and terminal compromise. The latter is a main concern of the study, possibly because it was written in a time where smart cards were simple containers of data that were read by the terminal, and the authors cope with it explicitly by including terminal verification in the studied protocols. In their model, the user and the smart card are different entities that share a secret (the PIN). The original logic is enhanced with a notion for secure channels and the notion of timely channel (a channel which was established recently), and this logic is being used to verify and prove correct delegation-based protocols, i.e., protocols that allow delegation of authority such as the user's card signing the terminal's credentials for certain time to authorize the terminal to act on his behalf.

Its applicability on protocols that do not involve delegation (i.e., most modern protocols) is limited.

2.2 The Shoup-Rubin game-based method

Shoup and Rubin proposed a game-based model to capture the usage of smart cards with symmetric keys in key distribution protocols involving a trusted party [28]. The threat model in [28] is a static corruptions model extended to model smart card theft, terminal tampering, etc. It is one of the first works to discuss the security implications and gains of using smart cards to protect cryptographic keys. The formal model used is similar to the Bellare and Rogaway model [5], which assumes a security parameter, a number of hosts n, and a trusted server S. Each host is given a smart card with long-term key $K_{1 \leq i \leq n}$ and S is given key K. The smart card is modeled as a stateless probabilistic oracle. On input x it returns $f(k, x)$. Each host i may communicate with a host

j by using a process $\mathcal{II}(i, j, u)$, where u is a process identifier to allow more than one connection. The adversary is a polynomial time probabilistic algorithm that initializes and interacts with the system. The interaction is a queries/answer based communication with the processes and the server. The "transcript" is a list that contains the queries and answers ordered in time. The allowed queries to a process are *delivery of a message, and response*, as well as three special queries (1) for a process to *reveal its session key*, and (2) for a smart card to *reveal its long-term key* and (3) to *access* an operation of the smart card oracle. If the latter special queries are used, the process or the smart card are considered to be *opened*. During the interaction each process may output a message indicating *acceptance* which indicates that a session key was established.

The Shoup-Rubin verification approach is used to verify three-party protocols using long-term symmetric keys. While its main game-based idea can be extended to accomodate two-party protocols as well, and even protocols using asymmetric primitives, it is not clear whether such an extension will result to an easy to use model that allows simple proofs.

2.3 Inductive verification

Bella uses the mathematical induction as the main tool to prove the protocol's security goals with respect to smart cards that protect symmetric keys [3]. The possible threats are modeled as a set of events, defined by inductive rules. The security goals are then proved using induction on the set. The an adversary model is similar to the one of Shoup-Rubin, with an additionally modeled threat which the author calls "data bus failure". This threat models smart card tampering in a way that messages from the card to the reader are modified or removed. The author then models the Shoup-Rubin protocol in [28] and uses the 'Isabelle' tool [22] to prove its security claims.

As with the Shoup-Rubin verification approach, inductive verification is used with long-term symmetric keys. An extension of it for asymmetric primitives would require a non-trivial modification of the model.

2.4 Resettable zero knowledge

Canetti et al. introduce the notion of resettable zero knowledge (rZK) [11]. This notion expresses an improvement over the classical notion of zero knowledge. Protocols under rZK remain secure even after a prover is reset to its initial state and re-uses the same random numbers. The resetting property in this notion is of particular importance to certain smart cards (e.g., smart cards that do not support atomic updates) due to their nature of being under the complete control of the adversary. A protocol that satisfies the rZK definitions is suitable for a cryptographic protocol that utilizes smart cards.

That notion while it may be interesting for the design of future protocols that target certain types of smart cards, it does not apply to any existing protocols that do not involve algorithms that satisfy strong notions such as (resettable) zero knowledge.

2.5 Our approach

In this paper we enhance Shoup's simulatability-based static corruptions model [27] to account for threats in a typical key establishment protocol arising from smart card usage. That is, we extend Shoup's model to incorporate the relevant to smart cards threats of the models above. This results in a model that can be used to verify any two-party key exchange protocol with symmetric or asymmetric keys.

Under this model we prove the modified PKINIT protocol secure. The notion of security in this model depends on an ideal system, with an ideal key exchange that is by definition secure and a real system that describes the actual protocol and participant interactions. A proof of security in this model shows that any attack on the real system can be simulated in the ideal system. We use this model because it is dedicated to key exchange, and, as such not only suffices for our purposes, but has the advantage of making this work relevant to all two-party secure communications protocols. Furthermore, due to its composition properties [27], it is a suitable tool to study protocol components in isolation.

3 Overview of the DH variant of PKINIT

Kerberos is an authentication and key distribution protocol originally proposed in 1988, and has a long history of attacks and updates (see, for example, [2, 12] and the references therein). Its main goal is to establish fresh session keys between users and servers and, as a result, enable users to log into multiple servers that belong to a common infrastructure. To this end, a user first requests an electronic 'ticket' from a central Key Distribution Center (KDC). The ticket then enables authenticated users to log into a server that is part of the infrastructure.

The main characteristic of all Kerberos variants, as well as the main difference with other Internet security protocols like TLS [15] or IPSec [18], is a single round-trip message exchange between the client and the KDC. This short message exchange is insufficient to provide any kind of freshness indication in the client message due to the lack of any prior input from the server, but it is handled by including a client-generated timestamp to ensure freshness (despite early arguments against that [6]). Replays in this exchange are prevented by requiring the server to store the client generated nonce during the validity time of the timestamp.

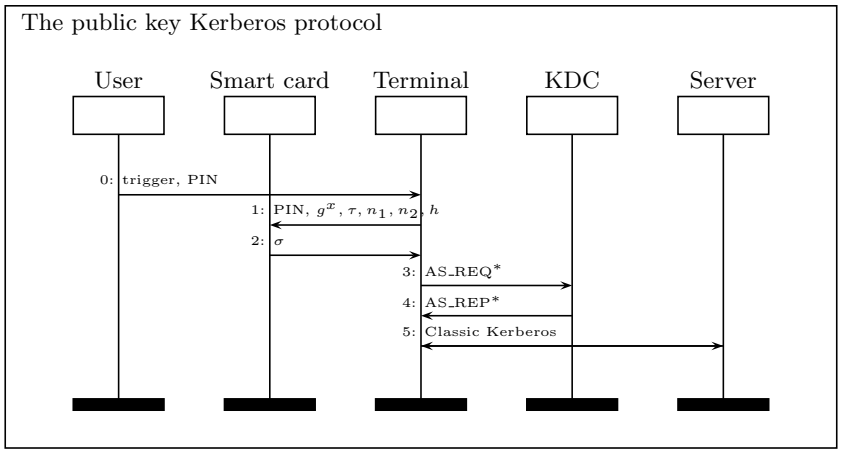

Figure 1: The DH variant of public key Kerberos with smart card (simplified).

Fig. 1 provides a high-level overview of the DH of PKINIT as specified in [21, 30] and specifically its variant where each party generates a fresh pair of DH keys. When the user decides to log into a server (step 0 in the Figure) his terminal constructs an `AS_REQ` message as specified in [21] and then computes its hash value h. Then it chooses a DH group, randomly generates an ephemeral DH secret $x \in \mathbb{Z}_p$, and computes g^x, where p is a large prime and g is the generator of the group. In that process the terminal chooses the 32-bit nonces n_1, n_2 and stores the current time τ. It then provides the values g^x, n_1, n_2, τ and h to the smart card for signing (step 1). Note that, depending on the implementation, it may only provide a hash of these values to the smart card. If the smart card is PIN-enabled, then the user must provide his PIN prior to this operation. Depending on the implementation, the PIN is inserted either to the terminal or the smart card reader. Fig. 1 shows the case where the user provides his PIN to the terminal in step 0, and where this is sent to the card in step 1.

The signature σ, output by the smart card (step 2), is then used by the terminal to construct an augmented version of the `AS_REQ` message, which we denote by `AS_REQ`*. This message, which contains $g^x, n_1, n_2, \tau, \sigma$ and `AS_REQ` as a substructure, is sent to the KDC (step 3) which, among other things, verifies the signature. If verification succeeds, then the KDC chooses a random $y \in \mathbb{Z}_p$, computes g^y and the ephemeral secret $K = F(g^{xy}, n_1)$, and constructs a response `AS_REP`. This message contains a ticket which is encrypted with K. Finally, it augments this message with fields containing the values of g^y and n_1 signed with server's private key. The resulting message, denoted `AS_REP`*, is sent to the terminal (step 4).

Using the value g^y and its ephemeral secret x, the terminal recovers the key K and is therefore able to decrypt the ticket. This ticket enables the terminal to complete the subsequent message exchange with the desired server (step 5).

The details of this exchange are not relevant to the attack. A more detailed view of the key exchange is shown below.

$$\begin{aligned}\text{User} \to \text{KDC} : \tau,\ \text{ID}_{\text{Server}},\ n_1,\ n_2,\ g,\ p,\ g^x,\ \text{cert}_{\text{User}},\\ \text{sig}_{\text{User}}(\tau, n_1, n_2, g, p, g^x, \text{ID}_{\text{KDC}})\\ \text{KDC} \to \text{User} : g^y,\ \text{ID}_{\text{User}},\ \text{cert}_{\text{KDC}},\ \text{sig}_{\text{KDC}}(g^y, n_1),\\ \{n_2, \text{ID}_{\text{Server}}, K_{\text{Server}}\}_K\end{aligned}$$

where $K = F(g^{xy}, n_1)$, F is a hash function based on SHA1 and K_{Server} is the key shared between the client and the server.

4 Attack and fix

This section revisits the attack on the PKINIT protocol as described in [19] and proposes a fix.

4.1 The attack

This protocol naturally lends itself to an implementation where the user's private signing key is stored in a smart card and indeed, PKINIT is typically used with smart cards in the Microsoft Windows Active Directory [14, 20]. However, in the combined protocol, there are three types of players: users, servers and a central entity called the Key Distribution Center (KDC). A user is equipped with a terminal and a smart card. The card contains an asymmetric key pair and a certificate, signed by an authority, that binds the public key to the user's identity. Moreover, the card provides an interface over which the terminal can ask the card to sign messages using the private key. Note that the card may require a PIN in order to respond to signature requests from the terminal.

Careful inspection of the smart card deployment documentation for Kerberos [14, 20] reveals a flaw, which arises due to the smart card introduction. This flaw leads to a relatively trivial attack, that enables an adversary with only temporary access to a victim's smart card, to impersonate the victim even after the adversary's access to the card is revoked. The attack is based on the observation that the KDC has no means to verify whether or not an incoming `AS_REQ`* message is fresh. That is, while the KDC checks that the timestamp τ indicates approximate current time, this does not guarantee that the `AS_REQ`* message was constructed recently. In fact, the `AS_REQ`* message could have been generated in the distant past.

In order to mount the attack, the adversary first obtains access to a victim's smart card. This can be done either by compromising the victim's terminal, or by stealing the card and its PIN. The adversary then fabricates an `AS_REQ` message, calculates its hash h, chooses nonces n_1, n_2, a random DH secret x,

and a timestamp τ indicating a particular future point in time. It then sends the values g^x, τ, n_1, n_2, and h to the card to obtain the signature σ. Using this signature, the adversary constructs an `AS_REQ`* message.

This fabricated `AS_REQ`* message will be accepted by the KDC as a genuine ticket request from the victim at time indicated by τ. Since neither the victim's himself, nor his smart card is required in the remainder of the protocol, the message enables the adversary to impersonate the victim to the KDC at time τ. With the ticket in the KDC' response, the adversary will further be able to impersonate the victim to the server of his choice until the ticket expires.

Note that, in order to be able to impersonate a victim at, say, approximately 20:00 of every Monday in a two-year period, the adversary must fabricate about 104 `AS_REQ`* messages as described above and, for each such message, obtain a signature from the card. In other words, a few minutes of access to a victim's card are sufficient for the adversary to be able to impersonate the victim, on a regular basis, for years.

4.2 The fix

A fix that defends against the attack at the cost of an additional message is described below. In this paper, we refer to the resulting fixed protocol as 'modified PKINIT'. In modified PKINIT, the KDC first sends a nonce n_0 to the terminal. This nonce is then added to the data signed by the smart card and included in `AS_REQ`* (see step 1 in Fig. 1). On reception of the `AS_REQ`* message, the KDC also ensures that the signature covers n_0. Apart from the message from the KDC to the client that transports the nonce, and a change to the `AS_REQ`* message to accommodate the additional nonce. The resulting protocol is shown below.

$$
\begin{aligned}
&\text{KDC} \rightarrow \text{User} : n_0 \\
&\text{User} \rightarrow \text{KDC} : \tau,\ \text{ID}_{\text{Server}},\ n_1,\ n_2,\ g,\ p\ , g^x,\ \text{cert}_{\text{User}}, \\
&\qquad\qquad \text{sig}_{\text{User}}(\tau, n_0, n_1, n_2, g, p, g^x, \text{ID}_{\text{KDC}}) \\
&\text{KDC} \rightarrow \text{User} : g^y,\ \text{cert}_{\text{KDC}},\ \text{sig}_{\text{KDC}}(\text{ID}_{\text{User}},\ g^x,\ g^y,\ n_1), \\
&\qquad\qquad \{n_2, \text{ID}_{\text{Server}}, K_{\text{Server}}\}_K
\end{aligned}
$$

where $K = F(g^{xy}, n_1)$ and F is a hash function based on SHA1.

Note however, that the fact that the protocol's initial message is sent by the server does not imply that this is no longer a client-initiated protocol. There are examples of client-initiated protocols where the server is sending the first session message, e.g., the SSH [29] protocol. Of course depending on the underlying transport layer an additional initial client message may be required to initiate the session.

An important note on the fix presented here, is the addition of g^x and $\mathrm{ID}_{\mathrm{User}}$ values to the signature value of the KDC. The original signature only contained the public parameter of the server and the client's nonce, making it applicable to a variety of sessions unrelated to this one that may share the same nonce (not an unlikely scenario given that this nonce is only 32-bits long). That weakness was first noted by Roy et al. [23] who nevertheless proved the security of the original scheme on the random oracle model, relying on the key derivation using the F function. However, the disassociation of a server-generated signature with the session that it was intended to be used, would in effect turn the server into a signing oracle, which is a known bad practice [1]. For this reason, we include these additional values in our modified version, a fact that also simplifies the study of the protocol and our later proofs.

4.3 Other possible fixes

The fix described above allows the re-use, for the purposes of Kerberos, of commonly available smart cards. These cards typically operate by signing any provided data or hash values by the terminal without interpreting them. On the other hand, the fix increases the protocol latency by an additional round-trip. If this is undesirable, alternative solutions may be possible at the cost, however, of requiring PKINIT-specific smart cards. That is, smart cards that interpret the data send by the terminal for signing. Examples are smart cards that would allow time to only go forward [25], effectively blocking the user's card if under the described attack, or smart cards with an internal clock that would set their time to the signed data. Another possible fix would be to obtain the initial nonce from the server through a layering violation. For example, in case of Kerberos over TCP, set the nonce to match the TCP sequence number used by the server.

These fixes, however, come at the cost of more expensive or specialized smart cards, or in the case of a layering violation, result in a protocol that can only be implemented in certain specific cases. For that, we will not consider them in this document any further.

5 A model for static corruptions with smart cards

This section describes the security model in which we analyze PKINIT and the fix. Our model is an extension of Shoup's model for session key establishment protocols [27] to the smart card setting. Note that we choose to extend the 'static corruptions' variant of Shoup's model, and that we omit certain options that are not relevant for our arguments in this paper. For a complete description of all variants of Shoup's model the reader is referred to [27].

The security model is based on simulatability of the real world within an ideal

world. That is, protocols that are secure in this model behave 'as if' the interactions between participates would take place in the ideal world. This ideal world is part of the model's definition and captures what it means for the protocol to be secure.

More specifically, security is defined by means of a 'game', i.e., a series of interactions, between an adversary and a 'ring master'. The purpose of this game, which takes place either in the 'real world' or in the 'ideal world', is the generation of a transcript. In the ideal world, the interactions between the adversary and the ring master cause protocol instances to be initialized, session keys to be generated and assigned to these instances, keys to be revealed, etc. However, in the ideal world, these protocol instances are 'virtual' because no security protocol is ever executed (and hence no protocol messages are ever exchanged); session keys are instead generated by the ring master, and information about these keys can be leaked to the adversary exclusively over well-defined interactions.

In the real world, the adversary also interacts with the ring master. In this world, however, the ring master controls 'real' protocol participants that execute the actual protocol. Session keys are indeed established by real protocol instances, and the only way for the adversary to drive these instances, is by means of sending protocol messages to the ring master. These messages are forwarded by the ring master to the protocol instances chosen by the adversary, and responses are forwarded to the adversary.

In both worlds, the adversary causes protocol events to occur, i.e., session keys to be established, sessions to be aborted, keys to be revealed, etc. These events are written to a transcript. The security definition requires that all real-world adversaries are 'simulatable' in the ideal world. That is, adversaries that interact, through the ring master, with real protocol participants, must not be able to cause a series of events that is impossible to be caused by an adversary that operates in ideal world which, by definition, is secure. Whether or not this 'simulatability' requirement is fulfilled, is verified by means of a 'distinguisher'. This distinguisher is given a transcript that originates from an adversary/ring master game, and must decide whether the game took place in the ideal or the real world. Only if the distinguisher has no advantage over random guessing is the protocol considered secure.

5.1 The ideal world

As explained above, in the ideal world, the adversary interacts with the ring master by means of different queries, defined below. During this interaction, the ring master creates state for 'users' and 'user instances', denoted by U_i and $I_{i,j}$, respectively, where $i, j \in \mathbb{N}$. The state associated with users and user instances is generated during this interaction. While the state associated with users models participants that may execute a given session key establishment protocol, the

state associated with user *instances* models individual executions the protocol. User instances must be 'initialized' before they can establish session keys. The ring master associates with every initialized instance a particular state; possible states are *active*, *isolated* and *finished*. The ring master may also reject certain queries; whenever this happens, the adversary has violated some consistency constraint which renders the particular query to be illegal.

The purpose of the adversary's interaction with the ring master is the generation of a transcript, i.e., a sequence of entries. Initially, the transcript is empty. The queries the adversary may issue during the game are as follows.

- *InitUser*(i, ID): On reception of this query where *ID* is a bit string, the ring master rejects the query if there exists an $(InitUser(i, \cdot))$ entry in the transcript. Otherwise, it assigns the identifier *ID* to U_i and appends the entry $(InitUser, i, ID)$ to the transcript. No information is returned to the adversary.
- $InitUserInstance(i, j, role, PID)$: On reception of this query where $role \in \{0, 1\}$ and *PID* is a bit string, the ring master rejects the query if the transcript does not contain any $(InitUser, i, \cdot)$ entry, or if it contains an $(InitUserInstance, i, j, \cdot, \cdot)$ entry. Otherwise, it sets the instance to be *active*, and assigns to it the role *role* and the partner identifier *PID* to the instance. This partner identifier is to be seen as the identifier of the user with which this instance shall expect to establish a session key. The ring master then appends the entry $(InitUserInstance, i, j, role, PID)$ to the transcript, and no information is returned to the adversary. Note that the role *role* breaks symmetry and can be seen as an indicator of whether the instance should behave as a client or server (equivalently, as an initiator or a responder).

Remark 1: Two instances may be 'compatible' with respect to a transcript. Informally, we say that instances are 'compatible' if they expect to establish a session key with each other. Formally, two instances $I_{i,j}$ and $I_{i',j'}$ are said to be *compatible* with respect to a transcript, if there exist entries $(InitUser, i, ID)$, $(InitUser, i', ID')$, $(InitUserInstance, i, j, role, PID)$, and $(InitUserInstance, i', j', role', PID')$ such that $role \neq role'$, $ID = PID'$, and $ID' = PID$.

- *AccessSC*(i): On reception of this query, the ring master rejects the query if the transcript does not contain any $(InitUser, i, \cdot)$ entry. Otherwise, it appends the entry $(AccessSC, i)$ to the transcript.

Remark 2: Note that *AccessSC*(i) is only a placeholder in the ideal world. The idea behind it is to simulate unauthorized access to smart cards, and its purpose is to enable the compromise mode in the *Start* query (see below).

- *Abort*(i, j): On reception of this query, the ring master rejects the query if there exists no $(InitUserInstance, i, j, \cdot, \cdot)$ entry in the transcript. Otherwise, it appends the entry $(Abort, i, j)$ to the transcript and sets $I_{i,j}$ to be *finished*. No information is returned to the adversary.

- *Start*$(i, j, mode[,key])$: On reception of this query, where $mode \in \{$*create*, *connect*(i', j'), *compromise*$\}$ and *key* is a bit string, the ring master rejects the query if there exists no $(InitUserInstance, i, j, \cdot, \cdot)$ entry in the transcript, or if $I_{i,j}$ is *finished*. Otherwise, it proceeds as follows.
  - If $mode =$ *create*, then the ring master generates a key $K_{i,j}$ uniformly at random, assigns it to $I_{i,j}$, sets $I_{i,j}$ to be *isolated*, and appends the entry $(Start, i, j)$ to the transcript. No information is returned to the adversary.
  - If $mode =$ *connect*(i', j'), then the ring master rejects the query if the two instances $I_{i,j}$ and $I_{i',j'}$ are not compatible. The ring master further rejects the query if the instance $I_{i',j'}$ is not *isolated*. The ring master then assigns the key $K_{i',j'}$, which was previously assigned to $I_{i',j'}$, to the instance $I_{i,j}$. Finally, the ring master sets both $I_{i,j}$ and $I_{i',j'}$ to be *finished*, and appends the entry $(Start, i, j)$ to the transcript. No information is returned to the adversary.
  - If $mode =$ *compromise*, then the adversary must also specify the parameter *key*. On reception of this query, the ring master rejects the query if there exists no $(InitUserInstance, i, j, \cdot, \cdot)$ entry in the transcript. Otherwise, it *selects* this entry. The ring master then checks whether or not $I_{i,j}$ has compatible instances. If it has, then the ring master checks whether or not the *first* $(InitUserInstance, i', j', \cdot, \cdot)$ entry, where $I_{i',j'}$ is a compatible instance, appears before the selected entry. If it does, then the ring master *deselects* the $(InitUserInstance, i, j, \cdot, PID)$ entry selected above and selects this $(InitUserInstance, i', j', \cdot, \cdot)$ entry instead. The ring master then checks whether or not the peer's the smart card was illegitimately accessed *after* the instance that was initialized first, was initialized. That is, the ring master checks whether or not the transcript contains an $(AccessSC, i')$ entry *after* the entry selected above. If *not*, then the ring master rejects the query.

    Finally, the ring master then checks whether or not the identifier *PID* corresponds to an initialized user, i.e., if there exists an $(InitUser, \cdot, ID')$ such that $PID = ID'$; if it does, then the ring master also rejects the query. If the query is not rejected, then the ring master assigns the key *key* to the instance $I_{i,j}$, sets the instance to be *finished*, and appends the entry $(Start, i, j)$ to the transcript. No information is returned to the adversary.

Remark 3: In simpler words the ring master accepts the *Start* with $mode =$ *compromise* query only when the peer is not an assigned user or if *AccessSC* has been issued on the peer after any of the current or the peer's instance have been initialized.

- *Application*(f): On reception of this query, where f is the description of a function that takes as parameters a string and a set of keys, the

ring master evaluates f on input the string R and the set of keys $\{K_{i,j}\}$ that have been assigned to instances during the game so far. The ring master appends the entry $(\mathit{Application}, f, f(R, \{K_{i,j}\}))$ to the transcript and returns the value $f(R, \{K_{i,j}\})$ to the adversary.

- $\mathit{Implementation}(\mathit{comment})$: On reception of this query, where *comment* is a bit string, the ring master appends the entry $(\mathit{Implementation}, \mathit{comment})$ to the transcript. No information is returned to the adversary.

Remark 4: To capture the threats of adversaries with temporary access to smart cards, the capabilities of the ideal world adversary are extended compared to the 'static corruptions' variant of Shoup's model [27]. The idea of our extension is that the adversary may explicitly access a participant's smart card by means of the *AccessSC* query. An adversary that causes an ongoing peer session to accept by doing so *after* the session was initialized, is considered benign in our model. This follows the intuition that, as long as an adversary has illegitimate access to a victim's smart card, it is no surprise if the victim can be impersonated. An adversary that causes a peer instance to accept without accessing the participant's smart card *while* their session is ongoing, on the other hand, this considered malicious. The distinction between these two adversary types is captured by the consistency constraints that the ring master enforces on reception of a $\mathit{Start}(\cdot, \cdot, \mathit{compromise}, \cdot)$ query. Note that modifying the constraints that apply to 'compromise' queries follows the general approach of the Shoup-Rubin model [28].

5.2 The real world

Unlike in the the ideal world, users and user instances are not simply placeholders in the real world. Instead, users correspond to real participants in a given session key establishment protocol, and are given long-term cryptographic keys and smart cards. User instances are probabilistic Turing machines that are activated for a particular execution of the protocol. As such, they accept messages and, for each incoming message, they output another message as well as a status indication from the following set.

- *continue*: The instance expects at least one more message.
- *accept*: The instance is finished and a session key has been established.
- *reject*: The user instance is finished without having established a session key.

Apart from users and user instances, in the real world there also exists a trusted third party (TTP). This TTP may operate off-line and has a public/private key pair. Users are required to register with this TTP before their instances can establish session keys. As in the ideal world, an adversary interacts with the ring master for the purposes of generating a transcript. The queries the adversary may issue to the ring master, are as follows.

- *InitUser*(i, ID): On reception of this query where *ID* is a bit string, the ring master rejects the query if there exists an $(InitUser, i, \cdot)$ or an $(Implementation, register, \cdot, \cdot, \cdot)$ entry in the transcript. Otherwise, U_i is registered with the TTP, his long term state is generated and his smart card is personalized and given to him. The ring master appends the entry (*InitUser*, i, *ID*) to the transcript, and no information is returned to the adversary.

- *Register*$(i, ID, request)$: On reception of this query where *ID* and *request* are bit strings, the ring master rejects the query if there exists an (*InitUser*, $i, \cdot$) or an $(Implementation, register, \cdot, \cdot, \cdot)$ entry in the transcript. Otherwise, it forwards *request* to the TTP, which registers the identity *ID*, and generates a response *response*. The ring master then appends the entry $(Implementation, register, ID, request, response)$ to the transcript, and returns `response` to the adversary.

- $InitUserInstance(i, j, role, PID)$: On reception of this query where $role \in \{0, 1\}$ and *PID* is a bit string, the ring master rejects the query if the transcript does not contain any $(InitUser, i, \cdot)$ entry, or if it contains an $(InitUserInstance, i, j, \cdot, \cdot)$ entry. Otherwise, it sets the instance to be *active*, assigns to it the role *role* and the partner identifier *PID* to the instance, and appends the entry $(InitUserInstance, i, j, role, PID)$ to the transcript. No information is returned to the adversary.

- *DeliverMessage*$(i, j, inMsg)$: On reception of this query where *inMsg* is a bit string, the ring master rejects the query if the transcript does not contain any $(InitUserInstance, i, j, \cdot, \cdot)$ entry. Otherwise, it sends the message *inMsg* to instance $I_{i,j}$. After processing this message, the instance outputs the message *outMsg* and the status τ. The ring master appends the entry $(Implementation, deliverMsg, i, j, inMsg, outMsg, \tau)$ to the transcript. If $\tau \neq continue$, the ring master sets $I_{i,j}$ to be *finished*. If $\tau = accept$, the ring master also appends the entry $(Start, i, j)$ to the transcript; if $\tau = reject$, it appends the entry $(Abort, i, j)$ instead. Finally, the ring master returns *outMsg* to the adversary.

- *AccessSC*$(i, request)$: On reception of this query where *request* is a bit string, the ring master rejects the query if the transcript does not contain any $(InitUser, i, \cdot)$ entry. Otherwise, it sends *request* to U_i's smart card. After processing this message, the smart card outputs the response *response*. The ring master appends the entries $(Implementation, accessSC, i, request, response)$ and $(AccessSC, i)$ to the transcript. Finally, the ring master returns *response* to the adversary.

- *Application*(f): On reception of this query, the ring master proceeds as in the ideal world, with the exception that the set of keys $\{K_{i,j}\}$ refer to the keys that were actually established by the protocol.

Note that there exists no *Implementation* query in the real world. We are now ready to state the soundness and security definitions.

Definition 1. *A smart card-based protocol is efficient and sound if real world user instances terminate after a polynomially bounded number of incoming messages, and, whenever the adversary faithfully delivers the generated messages between two compatible user instances, these instances accept and share the same session key.*
Definition 2. *A smart card-based protocol is secure if for every efficient real world adversary, there exists an ideal world adversary that generates a computationally indistinguishable transcript.*

Remark 5: In order to maintain the generality of our model, we did not introduce entities in the real world definition that are specific to PKINIT. While the key distribution server (KDC) is such an entity, it is handled just as another user that may also possess a smart card. While it is not mandatory that all users have smart cards, this approach naturally covers situations where the KDC stores its keys in a hardware security module (which operates similarly to a smart card).

Remark 6: Note that even though the PKINIT protocol has the notion of time and timestamps, they are not needed in the model. The "time" in the real or ideal world can be assumed to be given by a numeration of the transcript.

6 Results

This section presents our results, namely that PKINIT is not secure in the model described in Section 5, and that the fix proposed in [19] is. Our proof assumes that the Computational Diffie-Hellman (CDH) problem is hard, that the used signature scheme is unforgeable, and the random oracle model.[2]
Theorem 1. *The PKINIT key exchange with smart cards is not secure.*

Proof. This proof is an application of the distinguishing approach described in point 5 of section 3.4 in [27]. Consider the real world transcript shown in Fig. 2. According to entries 1 and 2, the adversary first initializes two users, 10 and 20, with identifiers 'Alice' and 'KDC', respectively. The adversary then accesses Alice's smart card in order to obtain the signature *signature* on the structure *signedData*. The adversary chooses *signedData* such that the concatenation $signedData||signature$ yields the fabricated `AS_REQ`* message described in Section 4.1. The adversary *afterwards* initializes an instance 200 for the KDC (see entry 5). Subsequently, the adversary sends the fabricated `AS_REQ`* message to KDC (see entry 6). Since this message is identical to a genuine message from Alice, the KDC accepts and establishes the session key $K_{20,200}$ (see entry 7). In entry 8, the adversary forces his guess of this session key into the transcript

[2]The following proofs can also be performed in the standard model, by assuming the Decisional Diffie-Hellman (DDH) problem is hard, and assuming that the function F represents a randomly selected function from a family of pair-wise independent hash functions (see [27, Section 5.3] for a discussion).

1. $(InitUser, 10, Alice)$
2. $(InitUser, 20, KDC)$
3. $(Implementation, accessSC, 10, signedData, signature)$
4. $(AccessSC, 10)$
5. $(InitUserInstance, 20, 200, 0, Alice)$
6. $(Implementation, deliverMsg, 20, 200, signedData||signature, msgFromKDC, accept)$
7. $(Start, 20, 200)$
8. $(Application, print\ \text{“guess”}, guess)$
9. $(Application, dump\ keys, \{K_{20,200}\})$

Figure 2: Example of a real world transcript without an indistinguishable ideal world equivalent

and then it asks the ring master to dump all established keys to the transcript. This results in the single established key $K_{20,200}$ to be written in entry 9.

We now describe a distinguisher that is able to determine, with non-negligible probability, whether a transcript of the above form originates from the real or the ideal world. The distinguisher simply indicates 'real world' if and only if the value of *guess* in entry 8 equals the session key $K_{i,j}$ reported in entry 9. This distinguisher is actually correct except with negligible probability. In order to see this, observe that, as explained in Section 4.1, the real world adversary knows the session key established by the KDC's instance. Therefore, with certainty, it holds that $guess = K_{i,j}$. There exist only two potential approaches to construct an ideal world transcript of this form where $guess = K_{i,j}$ holds; the first approach is by random guessing and this approach works with negligible probability. The other approach is to compromise the KDC's instance. Issuing a $Start(20, 200, compromise, key)$ query such that a $(Start, 20, 200)$ entry appears on position 8 in the transcript. This is, however, impossible for the ideal world adversary because this query is rejected unless either (a) the user identifier 'Alice' is not assigned to any user, or (b) Alice's smart card is accessed *after* the instance $I_{20,200}$, or any of its compatible instances, is initialized, and neither of these is true in this transcript. □

Theorem 2. *The modified PKINIT key exchange with smart cards is secure in the random oracle model, assuming that the Computational Diffie-Hellman (CDH) problem is hard with a security parameter k_d, the signature scheme is unforgeable with a security parameter k_s, signing keys exist only on smart cards, and nonces*[3] *are sufficiently long and random with a security parameter k_n.*

[3]It is, however, important to note that while the security parameters k_s and k_d are controlled by the user of the protocol by selecting longer key sizes and DH groups, the lengths of nonces are fixed to 32-bits.

Before proceeding with the proof, we restate the definition of the modified PKINIT protocol.

$$
\begin{aligned}
&U_{i'} \rightarrow U_i : n_0 \\
&U_i \rightarrow U_{i'} : \tau,\ \mathrm{ID}_{\mathrm{Server}},\ n_1,\ n_2,\ g,\ p,\ , g^x, \\
&\qquad\qquad \mathrm{cert}_i, \mathrm{sig}_i(\tau, n_0, n_1, n_2, g, p, g^x, \mathrm{ID}_{i'}) \\
&U_{i'} \rightarrow U_i : g^y,\ \mathrm{ID}_i,\ \mathrm{cert}_{i'},\ \mathrm{sig}_{i'}(\mathrm{ID_i},\ g^x,\ g^y, n_1), \\
&\qquad\qquad \{n_2, \mathrm{ID}_{\mathrm{Server}}, K_{\mathrm{Server}}\}_K
\end{aligned}
$$

Proof. We describe a simulator that transforms any real-world adversary into an ideal-world adversary such that no distinguisher has non-negligible advantage in determining whether a transcript originates from the interaction of the real-world adversary or our simulator.

Our simulator simulates real-world users and protocol instances towards the real-world adversary, and responds to queries of the form *InitUser*, *Register*, $InitUserInstance$, *DeliverMessage*, *AccessSC* and *Application* correspondingly. Towards the ideal-world ring master, our simulator behaves as follows. *InitUser*, *Register*, $InitUserInstance$, *AccessSC*, and *Application* queries are forwarded to the ideal-world ring master without the superfluous parameters. *Register*$(\cdot, \cdot, \cdot)$ queries are forwarded as $(Implementation, register, \cdot, \cdot, response)$ queries, where *response* is generated by our simulator. *DeliverMessage* queries are forwarded as $(Implementation, deliverMsg, i, \cdot, inMsg, outMsg, \tau)$ queries, where *inMsg* denotes the real-world adversary's message, and *outMsg* and τ denote the response message and status generated by the simulator's simulation of user U_i, respectively. Moreover, if an $(Abort, i, j)$ entry appears in the real-world transcript, our simulator issues an $Abort(i, j)$ query in the ideal world.

In order to cause the appearance of $(Start, i, j)$ entries in the correct positions of the ideal-world transcript while ensuring that the sets of established session keys remain computationally indistinguishable in both worlds (this requirement is important because otherwise a distinguisher can gain an advantage by examining the output of *Application* queries), our simulator proceeds as follows. Suppose a $(Start, i, j)$ entry appears on the real-world transcript, indicating that instance $I_{i,j}$ just received the last message it expected, accepted, and established a session key.

Case A: If $I_{i,j}$'s partner identifier *PID* does not correspond to an initialized user (if this holds in the real world transcript, it does so in the ideal world transcript as well), then our simulator issues a $Start(i, j, compromise, key)$ query, where *key* is the actual key established by $I_{i,j}$ in the real world. Note that this query results in the required $(Start, i, j)$ transcript entry. Moreover, it ensures that the key established by $I_{i,j}$ in the ideal world is identical to its real-world counterpart.

Case B: Otherwise, i.e., if there exists an $(InitUser, i', ID')$ entry in the transcript such that $ID' = PID$, then our simulator checks whether or not there

exists an $(AccessSC, i')$ entry in the transcript after the $(InitUserInstance, i, j,$ *role*, *PID*$)$ entry.

Case B.1: If such an entry exists, then our simulator issues a *Start*$(i, j,$ *compromise*, *key*$)$ query, where *key* is the actual key established by $I_{i,j}$ in the real world. this query results in the required $(Start, i, j)$ transcript entry. In effect, this means that the real-world adversary accessed the smart card after n_0 was generated and sent and thus is able to compromise the session.

Case B.2: Otherwise, i.e., if such an $(AccessSC, i')$ entry does not exist, then observe that, at least one instance $I_{i',j'}$ that is compatible with $I_{i,j}$ exists, except with negligible probability. That is, the nonce n_0 was sent before any "malicious" *AccessSC* by the adversary and the values n_1, g^x, g^y and the ID match. This follows from the structure of modified PKINIT, the assumption that the signature scheme is unforgeable, and the assumption that nonces are sufficiently long. Our simulator checks whether or not there exists an $(AccessSC, i')$ entry in the transcript after the $(InitUserInstance, i', j', role', PID')$ entry that corresponds to the *first* compatible peer instance $I_{i',j'}$.

Case B.2.1: If such an entry exists, then our simulator issues a *Start*$(i, j,$ *compromise*, *key*$)$ query, where *key* is the actual key established by $I_{i,j}$ in the real world. This results in the required $(Start, i, j)$ transcript entry.

Case B.2.2: Otherwise, i.e., if such an $(AccessSC, i')$ entry does not exist, then our simulator checks whether or not at least one of $I_{i,j}$'s compatible instances has accepted (and therefore are *isolated* in the ideal world).

Case B.2.2.1: If none of the compatible instances has accepted, then our simulator issues a *Start*$(i, j,$ *create*$)$ query. As a result, a random key $K_{i,j}$ is assigned to $I_{i,j}$ in the ideal world. Although this key is almost certainly not identical to its real world counterpart, it is computationally indistinguishable from it in the random oracle model (see section 5.3.3 in [27]).[4]

Case B.2.2.2: If at least one of the compatible instances has accepted, then our simulator selects one of these real world instances, denoted $I_{i',j'}$, by extracting the keys from all of them, and comparing it to the key established by $I_{i,j}$. If a match is found, then it issues a *Start*$(i, j,$ *connect*$(i', j'))$ query. Note that this results in the required $(Start, i, j)$ transcript entry, and causes the keys $K_{i,j}$ and $K_{i',j'}$ to be identical in the ideal world, just as they are in the real world. Our simulator therefore preserves the computational indistinguishability of the key sets and hence the transcripts. Note that, except with negligible probability, our simulator can always find a matching peer *isolated* instance and that this instance is unique. This follows from the construction of the PKINIT

[4]In the original PKINIT protocol, this assertion does not hold. In particular, using the attack described in Section 4.1, an adversary can cause $I_{i,j}$ to establish a key that is distinguishable from a random one; more precisely, the adversary knows the exact value of that key, as shown in Theorem 1. Moreover, since this adversary does not issue an *AccessSC*(i') query after the $(InitUser, i', ID)$, our simulator cannot simulate this adversary as this would require issuing an illegal *compromise* query.

protocol, the assumed security of the signatures, and assuming that nonces are sufficiently long. If, for example, the adversary had illegitimately accessed the peer's smart card, then its $AccessSC(i', \cdot)$ queries must have been issued *before* the $InitUserInstance(i', j', role', PID')$ query because otherwise our simulator would have branched to Case B.2.1. Hence, except with negligible probability, the adversary must have faithfully forwarded protocol messages between $I_{i,j}$ and $I_{i',j'}$ and that, as a result, their keys $K_{i,j}$ and $K_{i',j'}$ are identical. □

7 Conclusions

In this paper we extended Shoup's security model to include threats in the smart card setting, and examined the DH variant of the public key Kerberos protocol (PKINIT) in the new model. The new model incorporates smart card threats such as temporary access and terminal tampering, and is shown to capture the attack described in [19]. Furthermore, we show that our proposed fix for the PKINIT protocol is secure in that model.

The new model's applicability is not restricted to the Kerberos protocol and applies to any protocol that can be expressed in Shoup's model [27]. Given the broad scope, and also the simplicity behind the original model, we believe that the smart card version of the model would be a practical tool that can be used to verify other real world smart card-based protocols.

What the proof of security of the modified PKINIT protocol achieves, is to transform the intuition of security in the smart card setting due to the additional nonce, to a formal assurance of security. On the other hand, a formal assurance of security may not constitute a sufficient reason to introduce a protocol change, which is not trivial to deploy. We believe, however, that given the fact that main use-case of the PKINIT protocol in Microsoft Windows Active Directory is in combination with smart cards, the modification is a reasonable trade-off to ensure security in the smart card setting.

8 Acknowledgments

The authors would like to thank Alfredo Rial, Berry Schoenmakers and the anonymous referees for their comments which improved this manuscript. This work was supported in part by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen) SBO project, the Research Council KU Leuven: GOA TENSE (GOA/11/007), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

References

[1] R. J. Anderson and R. M. Needham. Robustness principles for public key protocols. In *Advances in Cryptology – CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 1995.

[2] M. Backes, I. Cervesato, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Cryptographically sound security proofs for basic and public-key Kerberos. *International Journal of Information Security*, 10(2):107–134, 2011.

[3] G. Bella. Inductive verification of smart card protocols. *Journal of Computer Security*, 11(1):87–132, 2003.

[4] M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. Cryptology ePrint Archive, Report 1998/009, 1998.

[5] M. Bellare and P. Rogaway. Provably secure session key distribution: the three party case. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 57–66. ACM, 1995.

[6] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. *Computer Communication Review*, 20:119–132, October 1990.

[7] S. Blake-Wilson, D. Johnson, and A. Menezes. Key agreement protocols and their security analysis. In *Crytography and Coding*, volume 1355 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 1997.

[8] B. Blanchet, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Computationally sound mechanized proofs for basic and public-key Kerberos. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 87–99, 2008.

[9] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on computer systems*, 8:18–36, 1990.

[10] M. Burrows, C. Kaufman, B. Lampson, M. Abadi, and M. Abadi. Authentication and delegation with smart-cards. In *Science of Computer Programming*, pages 326–345, 1992.

[11] R. Canetti, O. Goldreich, S. Goldwasser, and S. Micali. Resettable zero-knowledge (extended abstract). In *STOC*, pages 235–244. ACM, 2000.

[12] I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key Kerberos. *Information and Computation*, 206(2-4):402 – 424, 2008. Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA '06).

[13] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358, April 2007.

[14] J. de Clerq. Microsoft TechNet: Smart Cards, 2011. Available at: `http://technet.microsoft.com/en-us/library/dd277362.aspx`.

[15] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), 2008.

[16] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198 – 208, 1983.

[17] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society Press, 1990.

[18] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996 (Standards Track), Sept. 2010.

[19] N. Mavrogiannopoulos, A. Pashalidis, and B. Preneel. Security implications in Kerberos by the introduction of smart cards. In *Proceedings of the 7th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*. ACM, 2012.

[20] MIT Kerberos Consortium. PKINIT configuration, 2011. Available at: `http://k5wiki.kerberos.org/wiki/Pkinit_configuration`.

[21] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard), July 2005.

[22] L. C. Paulson. Isabelle: The next 700 theorem provers. *arXiv preprint cs/9301106*, 2000.

[23] A. Roy, A. Datta, and J. Mitchell. Formal proofs of cryptographic security of Diffie-Hellman-based protocols. In G. Barthe and C. Fournet, editors, *Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 312–329. Springer, 2008.

[24] B. Schneier and A. Shostack. Breaking up is hard to do: Modeling security threats for smart cards. In *First USENIX Symposium on Smart Cards*, 1999.

[25] B. Schoenmakers. Personal communication.

[26] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor. Encountering stronger password requirements: user attitudes and behaviors. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, pages 2:1–2:20. ACM, 2010.

[27] V. Shoup. On formal models for secure key exchange, 1999. IACR ePrint archive 1999/012.

[28] V. Shoup and A. D. Rubin. Session key distribution using smart cards. In *Advances in Cryptology – EUROCRYPT*, pages 321–331. Springer, 1996.

[29] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006.

[30] L. Zhu and B. Tung. Public Key Cryptography for Initial Authentication in Kerberos (PKINIT). RFC 4556 (Proposed Standard), June 2006.